# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. REPORT DATE (DD-MM-YYYY) | 2. REPORT DATE | 3. DATES COVERED: *(From – To)* |
|---|---|---|
| 01-04-2002 | FINAL | 09/23/96-09/23/00 |

**4. TITLE AND SUBTITLE**

A Federal Approach to Formal Hardware Design Verification

**5a. CONTRACT NUMBER**
DABT63-96-C-0097-P00006

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

David L. Dill

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Stanford University
651 Serra Street
**Stanford, CA 94305**

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Directorate of Contracting
ATZS-DKO-1
P.O. Box 12748
Ft Huachuca, AZ 85670-2748

**10. SPONSOR/MONITOR'S ACRONYM(S)**

**AGENCY REPORT NUMBER**

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for public release; distributution unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

20020411 082

**15. SUBJECT TERMS**

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | C. THIS PAGE | | | David L. Dill |
| U | U | U | UL | | 20b. TELEPHONE NUMBER (include area code) 650 725-3642 |

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI-Std Z39-18

# A Federated Approach to Formal Hardware Design Verification

## Final Report

## Administrative information

| | |
|---|---|
| ARPA order number: | E276 |
| Grant number: | DABT63-96-C-0097-P00006 |
| Actual Performance period: | 09/23/96-09/23/00 |
| Agent: | Dept of the Army, Ft Huachuca, Arizona 85670-2748 |
| Contract title: | "A Federated Approach to Formal Hardware Design Verification" |
| Organization: | Stanford University |
| Subcontractors: | SRI International |

## Introduction

The goal of this project was to enable the solution of hard formal verification problems – and thereby reduce the cost and improve the quality of advanced hardware designs – by making it possible to combine different verification techniques in flexible ways. This was be accomplished by developing an open software environment that allows various techniques to be combined easily. Various prototype tools were developed to support and use this environment, and results were evaluated on some example designs of interest.

There were significant accomplishments in several areas.

- We developed a better understanding of design and implementation issues in building a federated environment.
- SRI's widely-used PVS theorem prover was modified to provide APIs (application program interfaces) enabling it to be used in a variety of other tools. The enhanced version of PVS was used in the successful verification of hardware and safety-critical systems by other groups.
- We implemented a general-purpose system of cooperating decision procedures for quantifier-free first-order logic formulas, called SVC. This system integrated several decision procedures into a single framework, and was also used as a software component in several different verification tools, so it was "federated" at two different levels. The source code for the SVC system is freely available, and the system has been extensively used inside and outside of Stanford. Later, work was begun on re-engineering SVC to be more flexible, powerful, and efficient.
- We developed and prototyped new techniques for checking table-based specifications of embedded software systems, in the RSML description language. We analyzed and found some problems in an RSML specification of the safety-critical TCAS collision avoidance system.
- A tool for doing approximate symbolic model checking was prototyped and applied to an example design from another group at Stanford. Approximate model checking gains efficiency by computing an approximation of the reachable states of a design; however, the approximation is *conservative*, meaning that if the design is proved to have a given property using this technique, it actually *has* the property.
- The federated environment (including SVC) was applied to microprocessor verification. A new approach, called "incremental flushing" was developed for verifying more sophisticated architectures than previously. A proof of an out-of-order microprocessor design was done using this technique, using a combination of PVS and SVC.

## Implementation Issues

This project was based on the view that formal verification research would begin emphasizing combining tools and techniques to solve problems that were too difficult to handle using a single method. The vision of this project was that systems could be constructed from software components. Verification tools tend to be structured in three levels: low-level libraries for computation-intensive algorithms; higher-level algorithms and heuristics, which can have complex control structure, but are do not account for the bulk of computation time; and scripts or interactive commands that guide the application of the tool to examples.

More sophisticated systems could be constructed, and more challenging applications attempted, if we could make it much easier to combine techniques. Our strategy was to make use of efficient libraries in C and C++ for the low-level, computationally intensive code, and use another programming language (the "glue language") for the algorithmic code. Early in the project, we decided on the use of Common Lisp as the glue language for several reasons: it was already being used in the PVS implementation; it was well-suited for writing complex heuristic code, especially if it was not computation intensive; the availability of an interpreter meant that it could be used interactively; and that there was a smooth path from interactive commands to scripts to application tools.

Some libraries were available off-the-shelf, such as libraries for Boolean decision diagrams (BDDs), which are data structures for representing Boolean functions that are heavily used in formal verification applications. Other libraries, such as SVC (the Stanford Validity Checker), a decision procedure for quantifier-free first-order logic with uninterpreted functions, arrays, bit-vectors, and real linear arithmetic, were implemented during the project. In particular, SVC was written to be callable from other programs (rather than being a stand-alone tool) because of this project.

The choice of Common Lisp as the glue language was reasonably successful. Unfortunately, use for the language appears to be declining, but there are still good free implementations available for Linux. The students building prototype tools quickly learned the language, and generally found that, as planned, it made experimentation with the tools relatively easy. Although not all the work under the contract was based on the same implementation strategy, much of it was, and many of the sub-projects shared substantial amounts of code. Some of the microprocessor verification work and the approximate model checking sub-project were entirely implemented using this strategy.

In addition to proving the success of this approach, and enabling other research on the project to proceed more efficiently, we learned about some issues for implementing embeddable libraries. In the re-design of SVC, we have designed in a storage allocation method (based on reference counting) that should work better in embedded applications. We've also taken care to make the code re-entrant and re-initializable, because persistent global state in the library makes applications unpredictable and sometimes unreliable.

Similarly, SRI enhanced their widely-used PVS theorem prover to provide APIs that could be used in other tools. Since PVS is a large system, the process of exposing internal interfaces was done incrementally (indeed, this process is still underway). These enhancements received immediate use both inside and outside of SRI for verification of hardware and safety-critical systems. PVS also made better use of external components by calling them directly from Lisp instead of using intermediate files. This speeded up some verification tasks by a factor of a hundred.

One of the most important lessons was a bit surprising: well-designed and implemented libraries will work with almost any glue language. The design of component APIs is much more important than the choice of glue language. The API makes the library useful from almost any language that can call C or C++ functions, and the API does not have to change with the glue language.

## Decision procedures

Decision procedures for fragments of first-order logic have been a critical component in formal verification tools and related applications. Indeed, one of the most important attributes of the PVS theorem prover is that it incorporates decision procedures of various kinds, which can be used to deal with many tedious aspects of proofs automatically.

To enable powerful verication tools, we found that it was necessary to make more efficient and more general decision procedures. In particular, we found that reasoning about bit-vectors was critical for hardware

verification (and, it turns out, for other applications such as digital signal processing software), but that good decision procedures for bit-vectors were not available. We developed an efficient decision procedure for a theory of bit-vectors which included concatenation and extraction of fixed-width bit fields, as well as addition and subtraction. Our paper on this subject was awarded "best paper" at the Design Automation Conference in 1998 [1].

At a somewhat higher-level, it is desirable to build general decision procedures from individual theories (such as arrays or bit-vectors). Integrating diverse decision procedures so that the result gives correct results with reasonable efficiency is a non-trivial task. The SVC library that was produced as part of this project attempted to solve this problem, with at least partial success. Although SVC was begun under an earlier DARPA contract (NAG 2-891), and some of the development was done by students supported by other contracts, much of the development work, many optimizations and extensions, and the public release were funded by this project.

For various reasons, it has been difficult to benchmark the performance of SVC against the few other decision procedures of similar generality, but we are confident that, on most examples, it out-performs the others by orders of magnitude (authors of other packages have not disputed this claim). SVC has been the basis for a great deal of formal verification research at Stanford, and has also been used externally for verification of microprocessor designs, DSP software, and program analysis.

The source code for SVC is freely available on the web: `http://verify.stanford.edu/SVC`.

## Microprocessor Verification

This project included a great deal of work on the difficult problem of verifying microprocessor designs. The emphasis was to go beyond the very simple designs that were verified previously (e.g., simple RISC designs) to more realistic designs. The two primary directions were to attack more sophisticated architectural features, such as out-of-order instruction execution, and to attempt to verify actual designs created by other groups.

In previous work on verifying microprocessors, we discovered an automatic way to relate a pipelined microprocessor design to a non-pipelined formal specification of the "instruction set architecture" (ISA) of the microprocessor (the ISA is a programmer's view of what the machine does). This idea was extended at both SRI and Stanford to deal with more sophisticated architectures, which (for efficiency) can execute the instructions in a different order than the order specified by the program.

The original work on verifying pipelined processors was based on the concept of "flushing" the pipeline: the processor is controlled to complete any unfinished instructions, so that the resulting state is much easier to match up with the state of the ISA. The flushing method becomes computationally infeasible for the deep pipelines that exist in modern out-of-order architectures, so we developed a different method that flushes small numbers of instructions at a time [2, 3].

A great deal of effort was expended on trying to verify some designs created by Prof. Mark Horowitz's research group. We worked on two examples: Torch, which is a design created for research and education, and the protocol processor (PP) used in the FLASH multiprocessor design. Ultimately, we were not able to verify these designs formally, but we did develop some very useful techniques and prototype tools in the process. Several serious bugs were discovered in both designs that had escaped extensive previous verification efforts, including innovative methods that were the basis for a PhD thesis in Horowitz's group.

Tools that were developed for this work relied very heavily on the federated environment. We build a prototype Verilog translator and build a symbolic simulator for circuits, using Lisp and SVC. SVC was also used to answer queries about the results of symbolic simulation.

In both designs, we focussed on aspects of memory subsystems: the instruction fetch unit of Torch, and the memory interface unit of PP. We found that these designs were difficult to deal with using previous formal verification techniques, because they intimately combined control flow and data flow. Previous theorem proving methods would have had great difficulty with the complexity of the control flow in these designs. Previous methodology for model checking would have been to separate the control and data and abstract the data, but that approach fails for these circuits, because the desired behavior of the circuit cannot be specified for the control without the data flow.

Consequently, we developed a new verification method for such systems. The method requires isolating control finite state machines, identifying control states where the state of the data path is especially simple to

specify, then symbolically simulating the execution paths described by the regular expression. The method also includes heuristics for computing invariants around loops [4].

## Approximate model checking

Symbolic model checking using BDDs was (and continues to be) one of the more successful approaches to formal verification when this project started. However, the size of circuits that can be verified automatically is limited. This limitation is a major barrier to widespread practical application of model checking.

Model checking can be used on larger designs if we can settle for an approximate result. If such an approximation is consistently an over-approximation, it is still possible to conclude with certainty that a design has a desired property.

The core computation in model checkers finds the set of states that are reachable (for some inputs or internal nondeterministic choices) from a given initial state. Many correctness properties can be reduced to the question of whether all of these reachable states have a specified logical property. If it can be shown that every one of a *superset* of the reachable states has this property, it is guaranteed that all reachable states have the property (although, if a state in the superset fails to have a property, it cannot be known without further analysis whether said state represents a possible error or not, because it is not known whether the state is reachable).

In this project, we used a method of over-approximating the set of reachable states by *projecting* the sets onto subsets of the state variables. Intuitively, this can be thought of as finding an enclosing volume of a complex shape by looking at the shadows it casts in several directions. This idea had been explored earlier, but it required that the sets of variables onto which the states were projected had to be disjoint. We transcended this major limitation by finding a way to project the state space onto non-disjoint sets of variables. Some of the new manipulations on BDDs were discovered that will probably be more generally useful [5, 6, 7].

Several enhancements of this method were discovered later, including the idea of doing iterated forwards and backwards reachability of states (forwards from the start states, backwards from states violating a desired property). We also found an effective way to improve the choice of projection variables automatically. Using these ideas, we were able to verify circuits that we were not able pto do using exact model checking [8, 9].

## Specification checking

An important application that was prototyped using the federated environment was a checker for specifications of safety-critical software. We studied a particular specification, that of the TCAS air-traffic collision avoidance system. TCAS is a device that generates an audible warning if one plane is dangerously close to another. This is one of the largest and most complete specifications of a real safety-critical system that is publically available.

Specification languages, such as RSML, for safety-critical systems often insist that the specifications be *deterministic*, meaning that only one transition can be enabled at a time. However, the enabling conditions for transitions can be complex, so checking determinism can be difficult. The prototype tool used SVC to check for determinism of selected transitions in RSML specifications. Using the tool, we discovered a serious problem in the TCAS specification (it turned out that the problem had been discovered independently by a validation group, and fixed in a later revision of the specification) [10, 11].

## References


[1] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.

[2] Jens U. Skakkabæk, Robert B. Jones, and David L. Dill. Formal verification of out-of-order execution using incremental flushing. In *10th International Conference on Computer Aided Verification*, pages 98–109, June 1998.

[3] Robert B. Jones, Jens U. Skakkebæk, and David L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In Ganesh Gopalakrishnan and Phllip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *Lecture Notes in Computer Science*, pages 2–17, Palo Alto , California, November 1998. Springer-Verlag.

[4] Jeffrey X. Su, David L. Dill, and Jens U. Skakkebæk. Formally verifying data and control with weak reachability invariants. In *Formal Methods In Computer-Aided Design*, November 1998. Palo Alto, CA.

[5] Shankar G. Govindaraju, David L. Dill, Alan J. Hu, and Mark A. Horowitz. Approximate reachability with bdds using overlapping projections. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.

[6] Shankar G. Govindaraju and David L. Dill. Verification by approximate forward and backward reachability. In *Proceedings of International Conference on Computer-Aided Design*, November 1998. San Jose, CA.

[7] Gaurishankar Govindaraju. *Approximate Symbolic Model Checking Using Overlapping Projections.* PhD thesis, Stanford University, August 2000.

[8] Shankar G. Govindaraju, David L. Dill, and Jules P. Bergmann. Improved approximate reachability using auxiliary state variables. In *Proceedings of the 36th Design Automation Conference*, June 1999. New Orleans, LA.

[9] Shankar G. Govindaraju and David L. Dill. Approximate symbolic model checking using overlapping projections. In *First International Workshop on Symbolic Model Checking (SMC99) at Federated Logic Conference (FLOC)*, July 1999. Trento, Italy.

[10] David Y.W. Park, Jens U. Skakkebæk, Mats P.E. Heimdahl, Barbara J. Czerny, and David L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *FMSP'98: Second Workshop on Formal Methods in Software Practice*, pages 34–43, March 1998.

[11] David Y.W. Park, Jens U. Skakkebæk, and David L. Dill. Static analysis to identify invariants in RSML specifications. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 133–42, Lyngby, Denmark, September 1998.